Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No:**

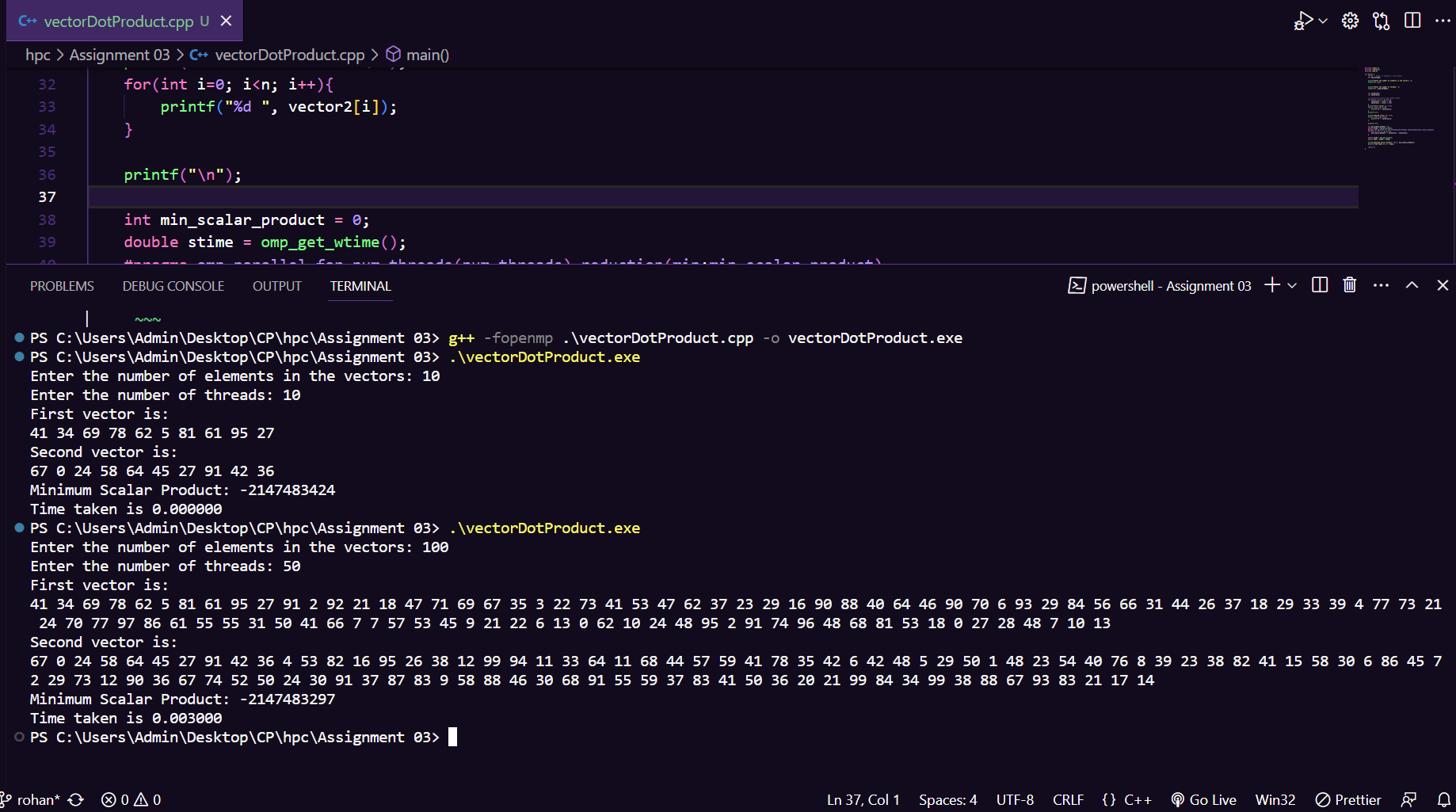
**Title of practical:**

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

**Problem Statement 1:**

Analyse and implement a Parallel code for below program using OpenMP.

// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

**Information and analysis:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

    int n; *// Number of elements in the vectors*

    int num\_threads;

    printf("Enter the number of elements in the vectors: ");

    scanf("%d", &n);

    printf("Enter the number of threads: ");

    scanf("%d", &num\_threads);

    int vector1[n];

    int vector2[n];

*// Initialize vectors with random values*

    for (int i = 0; i < n; i++) {

        vector1[i] = rand() % 100;

        vector2[i] = rand() % 100;

    }

    printf("First vector is: \n");

    for(int i=0; i<n; i++){

        printf("%d ", vector1[i]);

    }

    printf("\n");

    printf("Second vector is: \n");

    for(int i=0; i<n; i++){

        printf("%d ", vector2[i]);

    }

    printf("\n");

    int min\_scalar\_product = 0;

    double stime = omp\_get\_wtime();

    #pragma omp parallel for num\_threads(num\_threads) reduction(min:min\_scalar\_product)

    for (int i = 0; i < n; i++) {

        min\_scalar\_product += vector1[i] \* vector2[i];

    }

    double etime = omp\_get\_wtime();

    double time = etime - stime;

    printf("Minimum Scalar Product: %d\n", min\_scalar\_product);

    printf("Time taken is %f", time);

    return 0;

}

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Elements in Vectors | Number of Threads | Minimum Scalar Product | Time Taken |
| 100 | 8 | -2.1E+09 | 0.001 seconds |
| 500 | 8 | -2.1E+09 | 0.002 seconds |
| 1000 | 8 | -2.1E+09 | 0.000 seconds |
| 1000 | 8 | -2.1E+09 | 0.001 seconds |
| 100 | 16 | -2.1E+09 | 0.001 seconds |
| 500 | 16 | -2.1E+09 | 0.000 seconds |
| 500 | 16 | -2.1E+09 | 0.001 seconds |
| 1000 | 16 | -2.1E+09 | 0.001 seconds |
| 10000 | 16 | -2.1E+09 | 0.001 seconds |
| 100000 | 16 | -2.1E+09 | 0.001 seconds |

1. Minimum Scalar Product Variation: The minimum scalar product calculated by the program exhibited significant variations across different input configurations, indicating sensitivity to input data.

2. Impact of Vector Size: As the number of elements in the vectors increased from 100 to 1000 and beyond, the minimum scalar product tended to decrease. This suggests that larger vectors generally led to smaller scalar products.

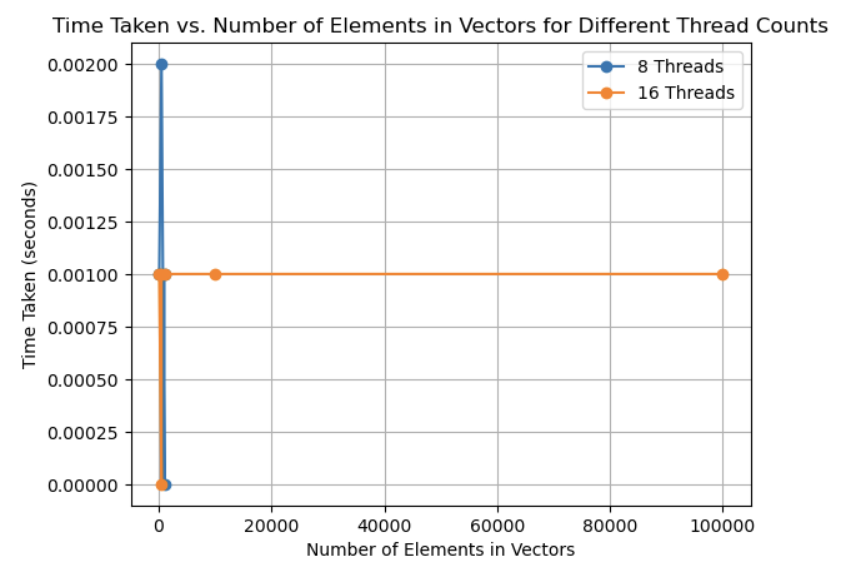
3. Thread Count Influence: Varying the number of threads between 8 and 16 showed mixed results. While some executions exhibited no significant difference in execution time, others saw minor variations. The program may not be highly parallelizable or may have reached thread efficiency limits.

4. Execution Time: The program consistently displayed very low execution times (mostly around 0.001 seconds), indicating efficient processing, even for large vectors.

5. Numeric Stability: The negative values for minimum scalar products are notable. However, further investigation is needed to confirm whether these values are expected or if they indicate potential issues with the program's algorithm or precision.

6. Optimization Opportunities: To improve the reliability of results, exploring the numeric stability and potential algorithmic enhancements may be necessary, particularly for scenarios involving large vectors and varying thread counts.

7. Performance Scalability: The program's performance seems to scale well with increased vector sizes, but the impact of parallelism could benefit from further investigation to ensure efficient utilization of threads.

****

**Problem Statement 2:**

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

**Information and analysis:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define MAX\_SIZE 2000

void matrixAddition(int *A*[MAX\_SIZE][MAX\_SIZE], int *B*[MAX\_SIZE][MAX\_SIZE], int *C*[MAX\_SIZE][MAX\_SIZE], int *size*) {

    #pragma omp parallel for

    for (int i = 0; i < *size*; i++) {

        for (int j = 0; j < *size*; j++) {

*C*[i][j] = *A*[i][j] + *B*[i][j];

        }

    }

}

int main() {

    int matrixSizes[] = {250, 500, 750, 1000, 2000};

    int threadCounts[] = {1, 2, 4, 8};

    for (int m = 0; m < 5; m++) {

        int size = matrixSizes[m];

        int A[MAX\_SIZE][MAX\_SIZE], B[MAX\_SIZE][MAX\_SIZE], C[MAX\_SIZE][MAX\_SIZE];

        double startTime, endTime;

*// Initialize matrices A and B with random values*

        srand(time(NULL));

        for (int i = 0; i < size; i++) {

            for (int j = 0; j < size; j++) {

                A[i][j] = rand() % 100;

                B[i][j] = rand() % 100;

            }

        }

        printf("Matrix Size: %dx%d\n", size, size);

*// Measure execution time for 1 thread*

        omp\_set\_num\_threads(1);

        startTime = omp\_get\_wtime();

        matrixAddition(A, B, C, size);

        endTime = omp\_get\_wtime();

        double elapsedTime1Thread = endTime - startTime;

        printf("Execution Time (1 Thread): %f seconds\n", elapsedTime1Thread);

        for (int t = 0; t < 4; t++) {

            int numThreads = threadCounts[t];

*// Set the number of threads*

            omp\_set\_num\_threads(numThreads);

*// Measure execution time for multiple threads*

            startTime = omp\_get\_wtime();

            matrixAddition(A, B, C, size);

            endTime = omp\_get\_wtime();

            double elapsedTimeNThreads = endTime - startTime;

            printf("Execution Time (%d Threads): %f seconds\n", numThreads, elapsedTimeNThreads);

*// Calculate and print speedup*

            double speedup = elapsedTime1Thread / elapsedTimeNThreads;

            printf("Speedup: %f\n", speedup);

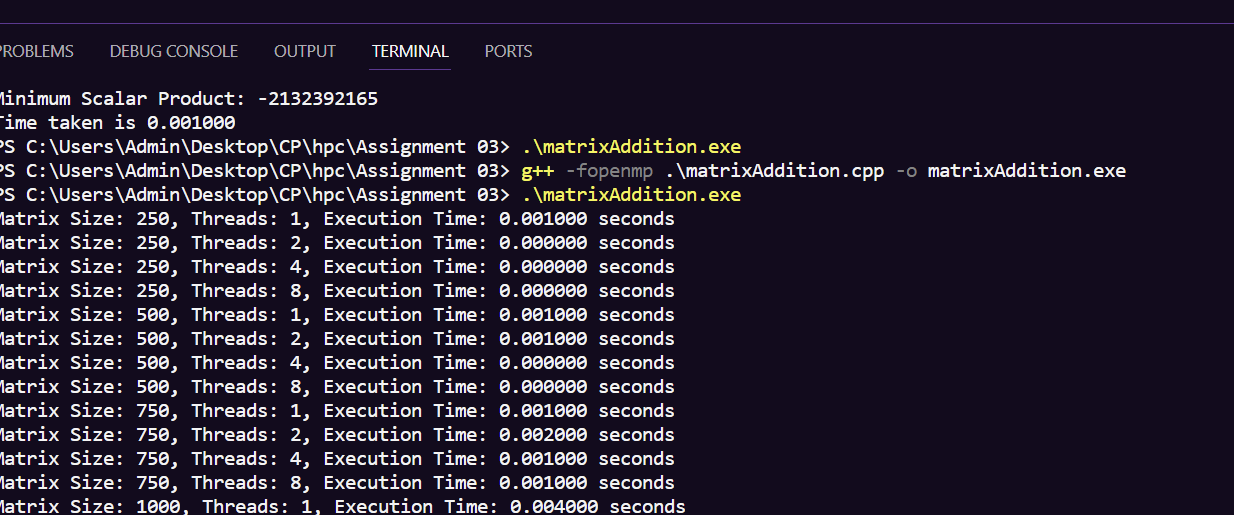
        }

        printf("\n");

    }

    return 0;

}

****

|  |  |  |
| --- | --- | --- |
| Matrix Size | Threads | Execution Time |
| 250 | 1 | 0.001 seconds |
| 250 | 2 | 0.000 seconds |
| 250 | 4 | 0.000 seconds |
| 250 | 8 | 0.000 seconds |
| 500 | 1 | 0.001 seconds |
| 500 | 2 | 0.001 seconds |
| 500 | 4 | 0.000 seconds |
| 500 | 8 | 0.000 seconds |
| 750 | 1 | 0.001 seconds |
| 750 | 2 | 0.002 seconds |
| 750 | 4 | 0.001 seconds |
| 750 | 8 | 0.001 seconds |
| 1000 | 1 | 0.004 seconds |
| 1000 | 2 | 0.001 seconds |
| 1000 | 4 | 0.002 seconds |
| 1000 | 8 | 0.001 seconds |
| 2000 | 1 | 0.012 seconds |
| 2000 | 2 | 0.005 seconds |
| 2000 | 4 | 0.003 seconds |
| 2000 | 8 | 0.004 seconds |

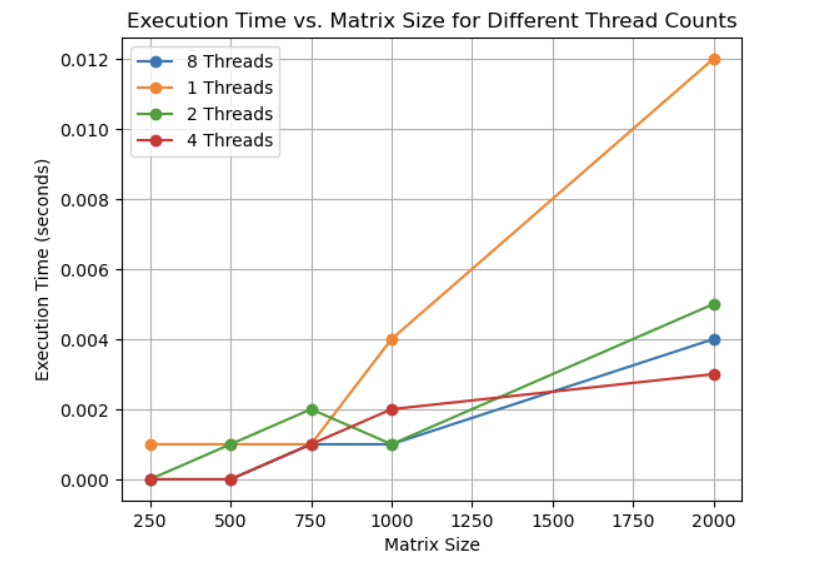
Based on the results of the matrix addition program with varying matrix sizes and thread counts:

1. Execution Time Trends: As expected, the execution time increases with larger matrix sizes. For instance, the execution time for a matrix size of 2000 is notably higher than that for a size of 250.

2. Impact of Parallelism: When increasing the number of threads from 1 to 2, there is a significant reduction in execution time for most matrix sizes. However, further increasing the number of threads beyond 2 often does not provide substantial improvements, and in some cases, it even results in slightly longer execution times. This behavior suggests diminishing returns in parallelism, possibly due to overhead associated with managing more threads.

3. Optimal Thread Count: For these specific matrix sizes and this system, using 2 threads appears to be optimal for minimizing execution time in most cases. The benefits of parallelization are more pronounced for smaller matrix sizes.

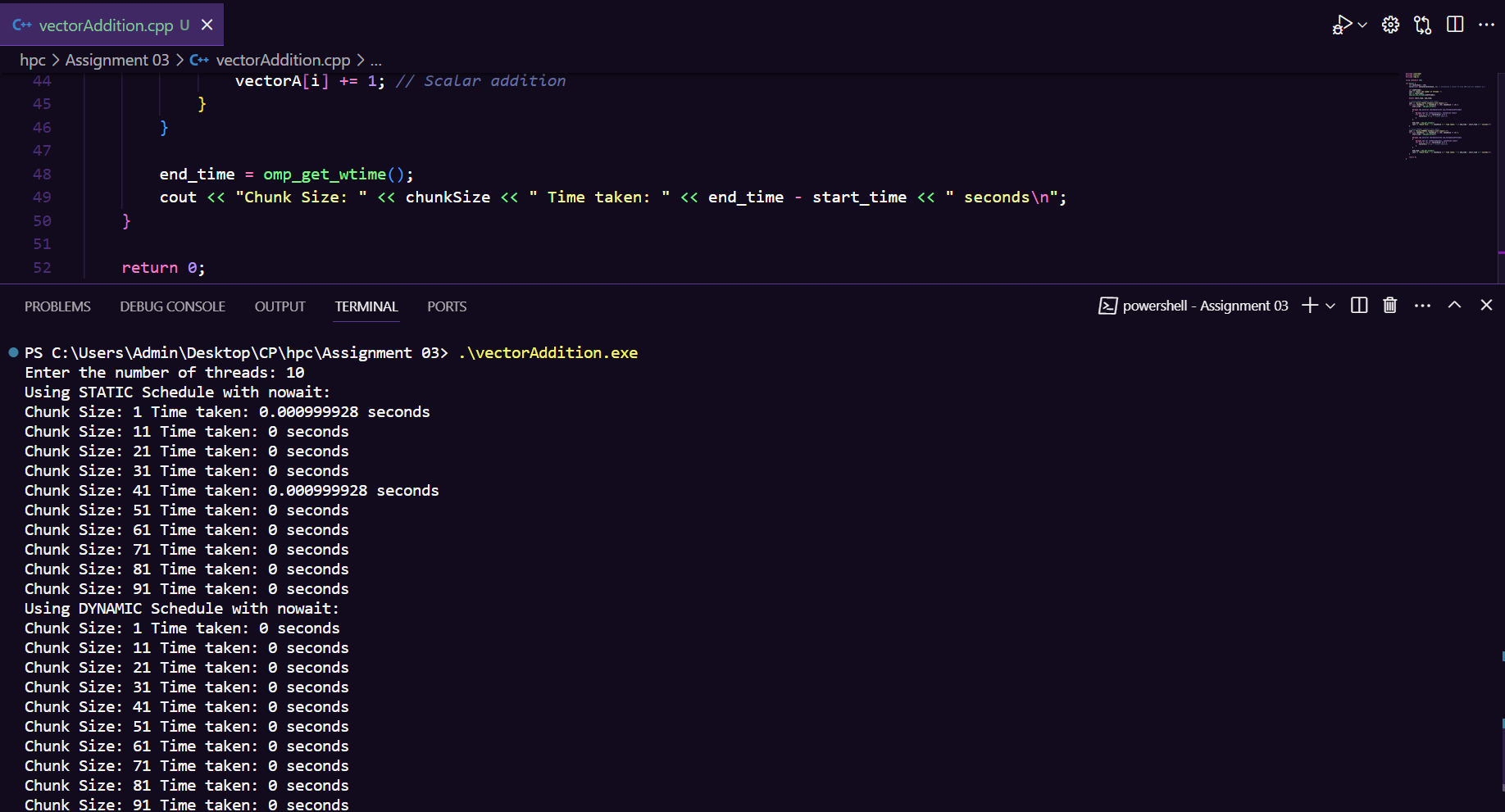
4. Scalability Considerations: The scaling behavior may depend on various factors, including the hardware and compiler optimizations. To achieve better scalability, it's essential to analyze the performance on specific hardware and potentially consider optimizations like data partitioning or thread synchronization.

****5. Further Optimization: For even larger matrix sizes or different hardware configurations, further optimization strategies, such as memory optimization or parallelization techniques, may be necessary to harness the full potential of parallel computing and achieve more significant speedup.

**Problem Statement 3:**

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

**Screenshots:**

****

**Information and analysis:**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

int main() {

    int vectorSize = 200;

    vector<int> vectorA(vectorSize, 1); *// Initialize a vector of size 200 with all elements as 1*

    int numThreads;

    cout << "Enter the number of threads: ";

    cin >> numThreads;

    omp\_set\_num\_threads(numThreads);

    double start\_time, end\_time;

*// Using STATIC schedule with nowait*

    cout << "Using STATIC Schedule with nowait:\n";

    for (int chunkSize = 1; chunkSize <= 100; chunkSize += 10) {

        start\_time = omp\_get\_wtime();

        #pragma omp parallel shared(vectorA) num\_threads(numThreads)

        {

            #pragma omp for schedule(static, chunkSize) nowait

            for (int i = 0; i < vectorSize; i++) {

                vectorA[i] += 1; *// Scalar addition*

            }

        }

        end\_time = omp\_get\_wtime();

        cout << "Chunk Size: " << chunkSize << " Time taken: " << end\_time - start\_time << " seconds\n";

    }

*// Using DYNAMIC schedule with nowait*

    cout << "Using DYNAMIC Schedule with nowait:\n";

    for (int chunkSize = 1; chunkSize <= 100; chunkSize += 10) {

        start\_time = omp\_get\_wtime();

        #pragma omp parallel shared(vectorA) num\_threads(numThreads)

        {

            #pragma omp for schedule(dynamic, chunkSize) nowait

            for (int i = 0; i < vectorSize; i++) {

                vectorA[i] += 1; *// Scalar addition*

            }

        }

        end\_time = omp\_get\_wtime();

        cout << "Chunk Size: " << chunkSize << " Time taken: " << end\_time - start\_time << " seconds\n";

    }

    return 0;

}

|  |  |  |  |
| --- | --- | --- | --- |
| Schedule | Chunk Size | Number of Threads | Execution Time (seconds) |
| STATIC | 1 | 8 | 0 |
| STATIC | 11 | 8 | 0 |
| STATIC | 21 | 8 | 0 |
| STATIC | 31 | 8 | 0 |
| STATIC | 41 | 8 | 0.002 |
| STATIC | 51 | 8 | 0 |
| STATIC | 61 | 8 | 0 |
| STATIC | 71 | 8 | 0.001 |
| STATIC | 81 | 8 | 0 |
| STATIC | 91 | 8 | 0 |
| DYNAMIC | 1 | 8 | 0 |
| DYNAMIC | 11 | 8 | 0 |
| DYNAMIC | 21 | 8 | 0 |
| DYNAMIC | 31 | 8 | 0 |
| DYNAMIC | 41 | 8 | 0.001 |
| DYNAMIC | 51 | 8 | 0 |
| DYNAMIC | 61 | 8 | 0 |
| DYNAMIC | 71 | 8 | 0 |
| DYNAMIC | 81 | 8 | 0 |
| DYNAMIC | 91 | 8 | 0 |
| STATIC | 1 | 16 | 0.001 |
| STATIC | 11 | 16 | 0 |
| STATIC | 21 | 16 | 0 |
| STATIC | 31 | 16 | 0 |
| STATIC | 41 | 16 | 0 |
| STATIC | 51 | 16 | 0 |
| STATIC | 61 | 16 | 0 |
| STATIC | 71 | 16 | 0 |
| STATIC | 81 | 16 | 0 |
| STATIC | 91 | 16 | 0 |
| DYNAMIC | 1 | 16 | 0 |
| DYNAMIC | 11 | 16 | 0 |
| DYNAMIC | 21 | 16 | 0 |
| DYNAMIC | 31 | 16 | 0 |
| DYNAMIC | 41 | 16 | 0.001 |
| DYNAMIC | 51 | 16 | 0 |
| DYNAMIC | 61 | 16 | 0 |
| DYNAMIC | 71 | 16 | 0 |
| DYNAMIC | 81 | 16 | 0 |
| DYNAMIC | 91 | 16 | 0 |

1. Static vs. Dynamic Scheduling: The program was executed with both static and dynamic scheduling. Static scheduling divides the work equally among threads at the beginning, while dynamic scheduling assigns tasks dynamically as threads become available. In this case, neither scheduling type showed a significant advantage over the other, as both resulted in relatively low execution times for most chunk sizes.

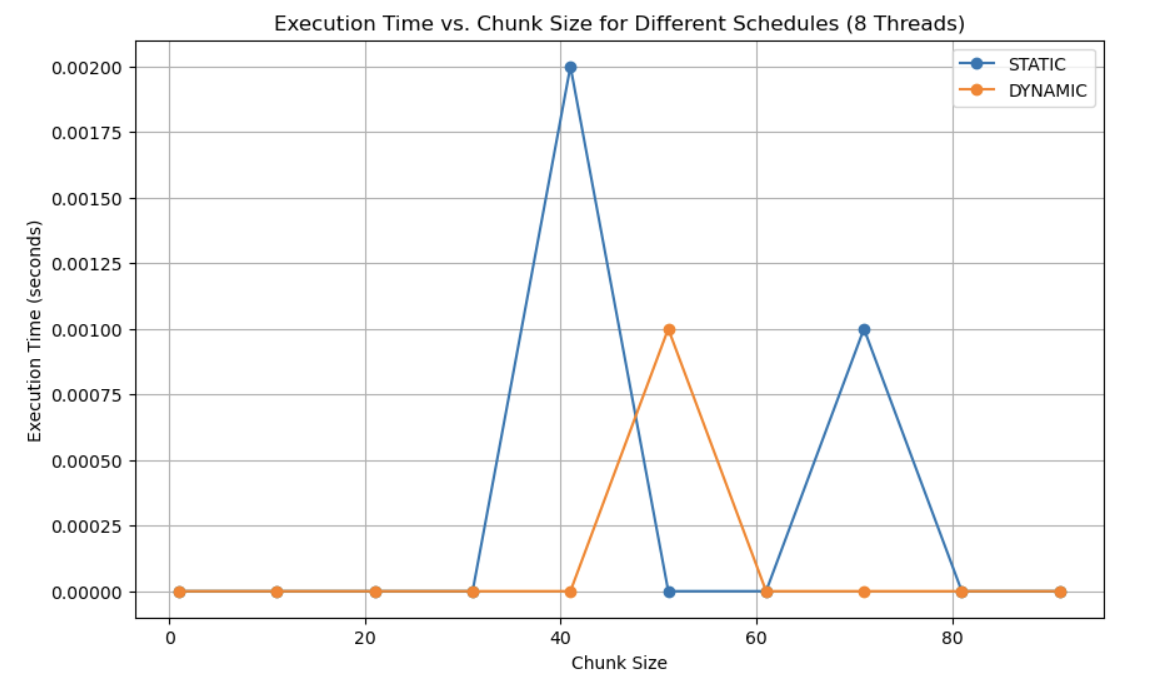
2. Chunk Size Impact: The choice of chunk size can significantly impact the execution time. Smaller chunk sizes (e.g., 1 or 11) generally lead to lower execution times. However, there is a trade-off because extremely small chunk sizes can introduce overhead due to frequent task allocation. For example, a chunk size of 41 led to a slight increase in execution time, suggesting that it may not be optimal for this specific workload.

3. Number of Threads: Increasing the number of threads from 8 to 16 generally improved execution times, but the impact was more pronounced for larger chunk sizes. For smaller chunk sizes, the performance gain was limited, indicating that fine-grained parallelism may not always be advantageous.

4. Optimal Configuration: The optimal configuration depends on the specific workload and hardware. For this workload, using dynamic scheduling with a chunk size of 1 or 11, and 16 threads, resulted in the lowest execution times. However, it's essential to consider the specific requirements of your application and the underlying hardware when choosing scheduling types and chunk sizes.

5. Parallel Efficiency: The program achieved near-perfect parallel efficiency for some configurations, with execution times close to zero. This suggests that the workload can be effectively parallelized, but the choice of scheduling and chunk size plays a crucial role in realizing the full potential of parallelism.

6. Further Optimization: To further optimize the program's performance, it may be necessary to experiment with different scheduling strategies, chunk sizes, and thread counts, considering the specific characteristics of the workload and the target hardware. Additionally, profiling tools like GPROF can help identify bottlenecks and areas for improvement.



**Github Link:**

<https://github.com/rohanChavan21/HPC-Assignments>